第3章 複数の小規模FFT回路を並列動作させて実現する64点FFT回路の設計

── Design Wave設計コンテスト2007 Student 部門第1位

チーム日本正彦(廣本正之，日向文彦)

Design Wave 設計コンテスト2007のStudent部門第1位の設計を紹介する．まず，回路規模を小さくすることを目標としてFFT回路を設計した．次に，スループットを上げるために，複数の小規模FFT回路を並列動作させるマルチコア化を行った．任意の並列度のRTLコードを自動生成するために，スクリプト言語を活用した．　(編集部)

私たちの所属している研究室では，システムLSIのアーキテクチャや設計方式などについて研究しています．研究の中でHDLを用いて回路を設計したり，FPGAに実装して動作させています．

研究室ではほぼ毎年，修士1回生が本コンテストに参加しています．そこで今年も先輩方に続き，沖縄における発表会を，さらにはコンテストでの優勝を目指したいと考えました．コンテストへの参加を通して，与えられた条件，期間内でハードウェアを設計する経験が得られると思い，課題の64点FFT回路の設計にチャレンジしました．

まず，回路規模の大半を占めるメモリを重点的に削減し，コンパクトなFFT回路を作りました．これを元に，スループットを拡張できるマルチコアのフレームワークを設計しました．ハードウェアの設計やFPGAへの実装，ボード上での組み込みシステムの実現などの貴重な経験ができました．

1. 設計方針を考える

まず，今回の設計で何をアピール・ポイントにするのか，何を狙ったアーキテクチャにするのかという所から考え始めました．

考えられるアプローチとしては，とにかく高いスループットを目指す，逆に回路面積をできるだけ小さくする，あるいはそのバランスの取れた設計を目指す，などがあります．通常の設計では，システムの要求仕様に応じてある程度方向性を決めることができます．しかし今回はコンテストということで自由度が高く，設計方針の決定には非常に頭を悩ませました．

当初はスループットもそこそこ高く，回路規模もそれなりに小さくしたバランスの良い設計を目指そうと思っていました．性能と面積のトレードオフを考慮した，王道とも言える設計方針です．しかし，これだと優れた回路はできますが，あまりおもしろくありません．せっかくコンテストに応募するので何か「オモロイ」ことをやろうと思い，ユニークな特徴を持たせた設計にしようと思いました．

● とにかく小さく

高いスループットか小さい面積か，どちらかにターゲットを絞って攻めてみるのがおもしろいのではないかと考えました．

ここで，FFTの応用例について簡単に考えてみました．設計仕様書にも書かれていた通り，64点FFTがよく用いられるのはOFDM(orthogonal frequency division multiplexing；直交周波数分割多重)方式などの無線通信用途です．このような用途においてはスループットも重要ですが，リソースの限られたモバイル機器上で動作させるため，小

KeyWord　FFT，マルチコア，RADIX-4，バタフライ演算，Design Compiler，FPGA，ML403，Virtex-4FX，スペクトラム・アナライザ

規模かつ低消費電力であることが求められます．そこで今回は，とにかく回路規模の小さなコンパクトなFFT回路を目指してみることにしました．

● まずは設計仕様書の回路を合成してみる

回路規模を削減すると言っても，演算器を小さくする，RAMやROMを減らすなど，複数の方法が考えられます．設計仕様書を読むだけではどこに着目すれば最も効果的に回路を小さくできるのか分からないため，まずは設計仕様書通りに回路を記述し，その合成結果を見て作戦を立てようと思いました．また，設計仕様書通りの実装例を作っておくことで，最終的に自分の設計と比較対象をすることもできます．

設計仕様書に従いRADIX-4のバタフライ演算器とシフト・レジスタを用いて設計した回路を**図1**に示します．各ステージはパイプライン動作し，シリアルに入力されるデータを絶え間なく処理することが可能です．リオーダ処理には64点データが格納可能なメモリを2セット用い，こちらも連続的にデータを流せるようにしました．回転因子については あらかじめ計算しておいた三角関数値を格納したROMを用いました．

この回路を米国Synopsys社のDesign Compilerを用いて合成し，遅延時間および回路面積の評価を行った結果を**表1**に示します．合成に用いたライブラリは台湾UMC（United Microelectronics Corp.）の0.18μmプロセスです．遅延時間の単位はns，回路規模は2入力NANDゲートに換算した場合のゲート数としています．**表1**にはFFT回路全体の合成結果のほかに，組み合わせ回路の中で大部分を占めるバタフライ演算器と回転因子乗算器についても，個別に合成した結果を示しています．演算器の合計規模は約9,000ゲートであり，FFT回路全体の約15%にしか過ぎ

表1　設計仕様書に従ったアーキテクチャの合成結果

	遅延時間［ns］	面積［ゲート］
FFT全体	4.98	62111
バタフライ演算器（ステージ1）	2.80	808
バタフライ演算器（ステージ2）	2.49	1061
バタフライ演算器（ステージ3）	2.75	1240
回転因子乗算器（ステージ1）	4.09	2928
回転因子乗算器（ステージ2）	4.29	3251

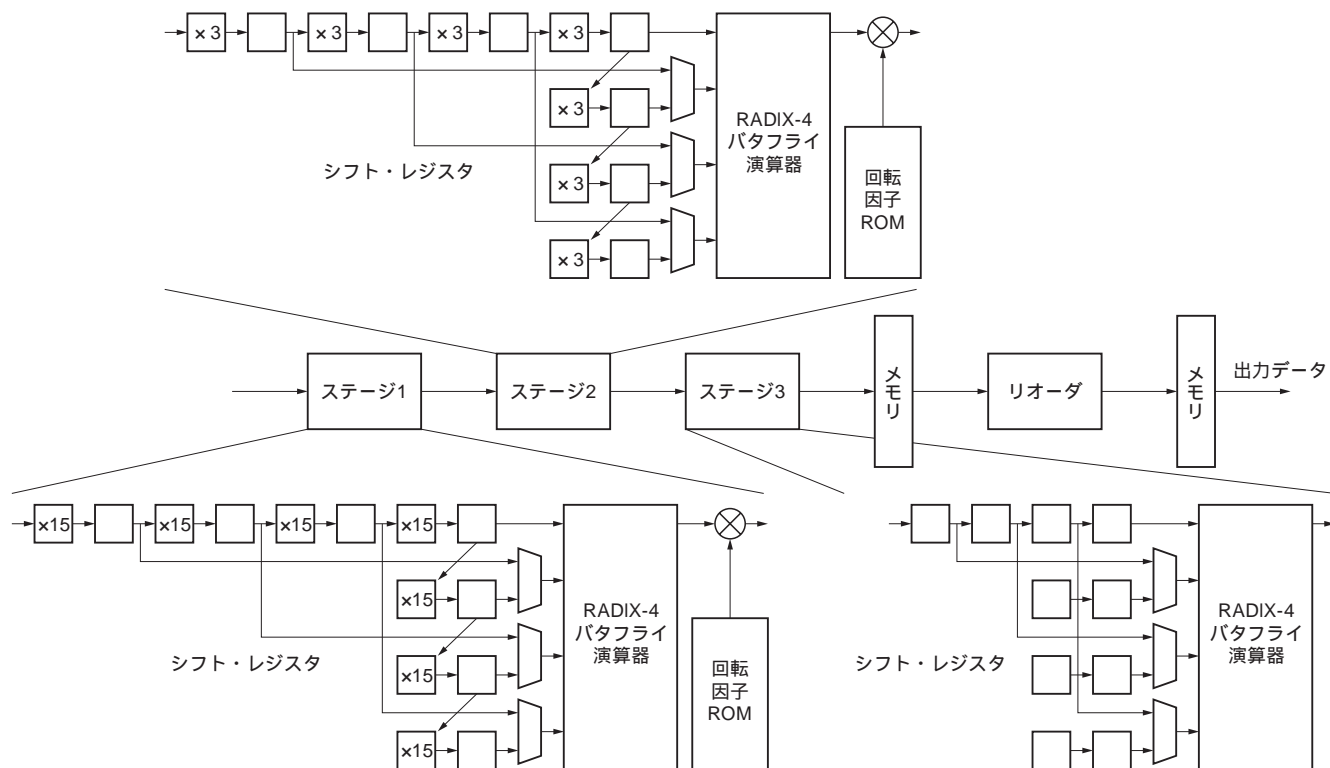



図1　設計仕様書に従ったアーキテクチャ

各ステージはパイプライン動作し，シリアルに入力されるデータを絶え間なく処理することが可能．リオーダ処理には64点データが格納可能なメモリを2セット用い，こちらも連続的にデータを流せるようにした．回転因子についてはあらかじめ計算しておいた三角関数値を格納したROMを用いた．

ません．回路面積の大部分はシフト・レジスタやRAMなどで占められていることが分かりました．

これらの結果より，回路の大部分を占めるシフト・レジスタやRAMといった記憶素子に焦点を当て，データの取り回しを工夫してメモリを削減することが全体の回路規模を小さくするのに効果的である，という結論に達しました．

2. 最小構成のFFT回路

FFTを行うためには64点のデータが格納できる64ワードのメモリが最低限必要です．今回はこの最低限のメモリのみを搭載することを考えました．また，演算器についてもRADIX-4バタフライ演算器を一つだけ使用することにしました．

アーキテクチャの概略図を**図2**に示します．記憶素子は64ワードのメモリが1個だけです．このメモリから必要なデータを順次読み出し，バタフライ演算器，回転因子乗算器でパイプライン的に演算を行い，演算後のデータは再びメモリに格納されます．このアーキテクチャでは単一メモリにより演算を行うため，入力データは一定の間隔を置いて連続した64点が入力されるものとしています．

● 全体動作

本アーキテクチャでFFTを行うようすを説明します．全体動作のタイミング・チャートを**図3**に示します．

最初の状態DATAINでは，入力される64点のデータを順次メモリに格納していきます．DATAIN状態の最後の4クロックでは，1ステージ目のRADIX-4バタフライ演算に必要な4データをロードします．

次状態のSTAGE1では，同時に演算を行い，結果をメモリに格納します．以降64サイクルかけて64点のデータを順に計算し，1ステージ目の処理を終えます．

その後メモリへのアクセス・パターンを変え，同様に64サイクルずつかけて2ステージ，3ステージの処理を行います．3ステージ目のデータ・ロードが終わり次第，リオーダ順にメモリからデータをロードし，FFTモジュールから出力を行います．

このアーキテクチャのスループットは1データ/4サイクル，レイテンシは64×4－4＝252サイクルです．

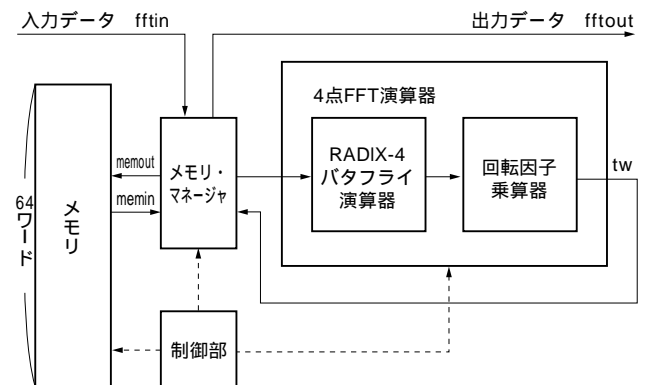



図2 小規模化を目指したFFT回路のアーキテクチャ
記憶素子は64ワードのメモリが1個だけ．このメモリから必要なデータを順次読み出し，バタフライ演算器，回転因子乗算器でパイプライン的に演算を行い，演算後のデータは再びメモリに格納する．

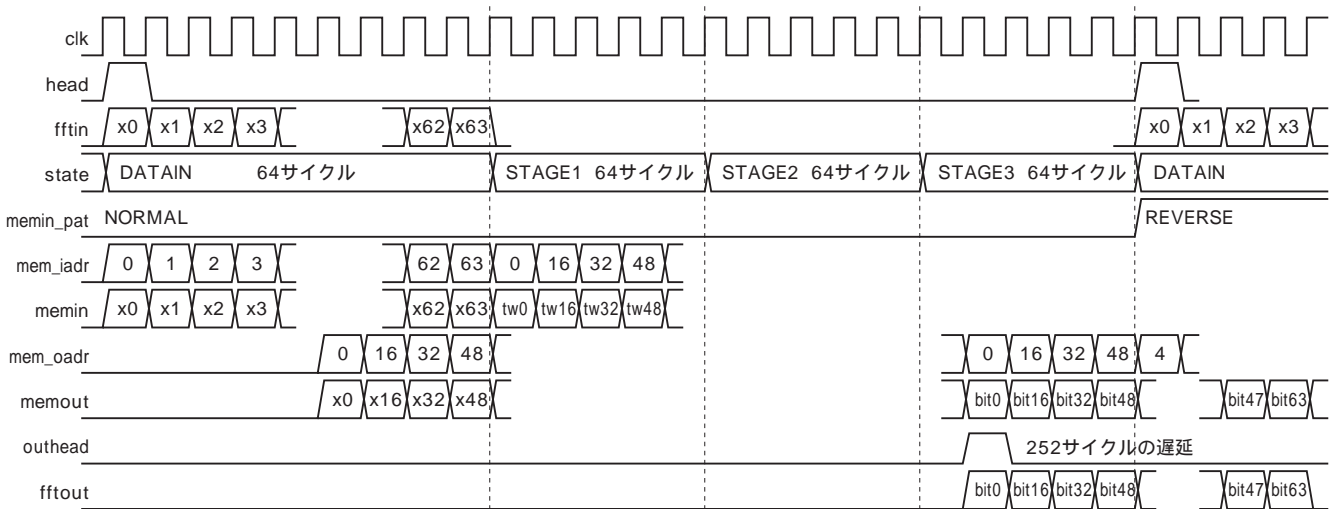



図3 小規模化を目指したFFT回路のタイミング・チャート
スループットは1データ/4サイクル，レイテンシは64×4－4＝252サイクル．

● メモリ

FFTを効率良く行うためにメモリ・モジュールに持たせた特徴は以下の2点です．

- 入力ポート，出力ポートをそれぞれ独立に用意
- リオーダ処理を効率化する2種類のアクセス・モード

1点目については，データの読み出し，書き込みがそれぞれ独立して行えるよう，入出力ポートをそれぞれ一つずつ用意しました．これにより，必要なデータをロードしつつパイプライン的に演算器に流し込み，結果をメモリに格納することが可能となります．これで1ステージにつき無駄なく64クロックで演算を終えることができました．

2点目については，メモリへのアクセス・パターンを2種類用意することにより，リオーダ処理に要するサイクルを隠ぺいすることに成功しました．ステージ3の終了後，メモリに格納されているFFTの演算結果をリオーダ順に出力する際，アドレス$x = k_0 + 4k_1 + 16k_2$のデータは，$y = 16k_0 + 4k_1 + k_2$番目に出力する必要があります．そのため，単純に**図3**のようにリオーダ順に出力しつつ次の64点データを入力するとメモリの上書きが生じ，演算結果が正しく出力されなくなってしまいます．

これを防ぐためには，リオーダ用のメモリをもう一つ用意するか，リオーダが終了してから次データを入力するかが考えられます．しかし前者は回路規模の増加に，後者はスループットの低下につながります．

そこで本設計ではリオーダ処理の特性に着目し，メモリを増やすことなく**図3**のタイミング通りにリオーダを行う手法を考えました．

リオーダ処理に相当する関数を，

$$R(x) = k_2 + 4k_1 + 16k_0 \quad (x = k_0 + 4k_1 + 16k_2,\ 0 \leqq k_0, k_1, k_2 \leqq 3) \qquad (1)$$

と定義します．ステージ3の終了後，リオーダ順にデータを出力するのでメモリは，$R(i)$＝{0, 16, 32, 48, 4, …}の順にアクセスされます．この順で出力しつつデータを潰さずに次のデータを書き込むには，データ入力の順もリオーダ順$R(i)$＝{0, 16, 32, 48, 4, …}とすればよいのです．つまり，次の入力データ{x_0, x_1, x_2, x_3, x_4, …}がアドレス{0, 16, 32, 48, 4, …}に格納されていくことになります(**図4**)．このようなメモリへのアクセス・モードを「REVERSEモード」と呼ぶことにします．なお通常のアクセス・モードは「NORMALモード」とします．

図4
REVERSEモードにおけるメモリへのデータ格納順序
つまり，次の入力データ{x_0, x_1, x_2, x_3, x_4, …}がアドレス{0, 16, 32, 48, 4, …}に格納されていく．

いったんREVERSEモードに入ると，後はすべてのメモリ・アクセスの際に式(1)に従いリオーダ順のアドレスに変換するだけで，正しく演算を行うことができます．さらに，リオーダ関数$R(x)$は2回作用させると，

$$R(R(x)) = R(k_2 + 4k_1 + 16k_0) = k_0 + 4k_1 + 16k_2 = x \qquad (2)$$

となり，元に戻る性質があります．よってREVERSEモードのリオーダ出力はメモリ・アドレス順の{0, 1, 2, 3, 4, …}となり，次データの入力順も元通り{0, 1, 2, 3, 4, …}となります．従ってREVERSEモードとNORMALモードを交互に繰り返すことによって，問題なく演算し続けることができます．

● バタフライ演算器

本設計で採用したバタフライ演算器は，基本的には設計仕様書と同じものです．特徴としては，

- 乗算回数の少ないRADIX-4バタフライ演算器
- シフト・レジスタによる連続4点のパイプライン演算

が挙げられます．回路構成を**図5**に示します．

● 回転因子乗算器

このモジュールは回転因子ROMと符号付き乗算器からなります．主な特徴は以下の通りです．

- 使用する値のみを回転因子ROMに格納
- ステージ1とステージ2でROMを共用

図5　4点FFT演算回路
設計仕様書と同じである．

	k_0 0	1	2	3
n_0+4n_1 0	W_{64}^{0}	W_{64}^{0}	W_{64}^{0}	W_{64}^{0}
1	W_{64}^{0}	W_{64}^{1}	W_{64}^{2}	W_{64}^{3}
2	W_{64}^{0}	W_{64}^{2}	W_{64}^{4}	W_{64}^{6}
⋮	⋮	⋮	⋮	⋮
15	W_{64}^{0}	W_{64}^{15}	W_{64}^{30}	W_{64}^{45}

(**a**) ステージ1

	k_1 0	1	2	3
n_0 0	W_{64}^{0}	W_{64}^{0}	W_{64}^{0}	W_{64}^{0}
1	W_{64}^{0}	W_{64}^{4}	W_{64}^{8}	W_{64}^{12}
2	W_{64}^{0}	W_{64}^{8}	W_{64}^{4}	W_{64}^{24}
3	W_{64}^{0}	W_{64}^{12}	W_{64}^{24}	W_{64}^{36}

(**b**) ステージ2

図6　回転因子
回転因子として0乗から63乗のすべての値を用いる訳ではない．

回転因子は，

$$W_{64} = \exp\left(-j\left(\frac{2\pi}{64}\right)\right) \cdots\cdots\cdots\cdots (3)$$

で与えられるW_{64}の0乗から63乗の値を取ります．しかし実際に回転因子として使われる値はステージ1で，

$$W_{64}^{k_0(n_0+4n_1)} \quad (0 \leqq k_0,\ n_0,\ n_1 \leqq 3) \cdots\cdots\cdots (4)$$

ステージ2で，

$$W_{64}^{k_1 n_0} = W_{64}^{4k_1 n_0} \quad (0 \leqq k_1,\ n_0 \leqq 3) \cdots\cdots\cdots (5)$$

であり，0乗から63乗のすべての値を用いる訳ではありません(**図6**)．使用しない値をROMに格納するのは回路がむだになるので，上式で使用する値のみをROMに格納することにしました．また，式(4)と式(5)から分かるように，ステージ2で使う値はすべてステージ1で使う値に含まれます．そこでステージ2用に新たにROMを設けることはせずに，ステージ1のROMを共用し，アクセス・パターンを変えることによりステージ2用の値を得ることにしました．

● 回路の評価

以上のようなFFT回路をHDLで記述し，Design Compilerにより遅延時間と回路規模を評価しました．合成結果を**表2**に示します．実験条件は仕様書に従って設計したときと同じです．

遅延時間についてはやや増加していますが，回路規模が設計仕様書の約40%にまで減少しており，メモリ削減の効果が確認できます．各モジュールの内訳を見るとまだメモリの比率が高く，全体の回路規模削減のためにメモリに着目したことが有効であったことが分かります．

表2　小規模化を目指したFFT回路の合成結果

		遅延時間［ns］	回路規模［ゲート］
仕様書の回路		4.98	62111
設計した回路	FFT全体	5.97	24708
	メモリ	0.48	16823
	シフト・レジスタ	0.14	1315
	バタフライ演算器	2.75	1240
	回転因子乗算器	5.06	4063

3．マルチコアFFT回路

コンパクトなFFTモジュールとして，最低限のメモリを搭載した回路を設計しましたが，これだけでは少しおもしろみが足りません．この小さなFFT回路を元に，何かもう一工夫できないかと考えました．

とはいえ，実質的な作業に取り掛かったのが年が明けてからだったので，締め切りの1月26日まであまり時間がありませんでした．ここまでの時点で既にあと2週間．厳しくなってきました．設計だけでなくFPGA実装までぜひやりたいと思っていたので，限られた時間内でできるだけ効果的でおもしろいアイデアがないか，頭を捻る日々でした．

● フレキシブルなFFT回路に

これまで設計してきた回路は，スループットをやや犠牲にしつつ，できるだけ小さな回路を目指したものでした．コンパクトなモジュールは，ほかの機能といっしょにシステムへ組み込む際には非常に重宝すると思われます．しかし，用途によってはもう少し回路が大きくなってもいいからスループットが欲しい，という要求があるかもしれません．この場合，いちいち要求性能に応じた回路を設計していたのでは効率が悪く，あまりうれしくありません．そこで，設計したコンパクトなFFTモジュールをベースに，さまざまなスループットに対応できる柔軟性の高いアーキテ

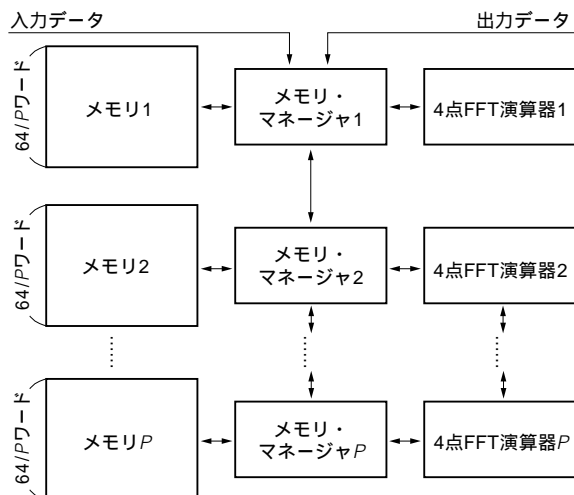



図7　マルチコアFFT回路のアーキテクチャ
並列度Pの場合，メモリがP分割されてP個の演算器に割り当てられ，それぞれがメモリ分散型マルチプロセッサのように動作する．

図8　2コアFFT回路

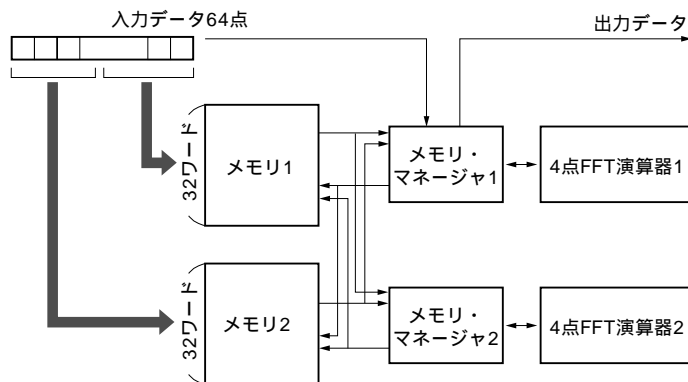
クチャにしてはどうかと考えました．必要なスループットを与えると，それに応じたFFTモジュールが生成されるフレームワークがあるときっと便利だろうと思いました．

思い立ったら早速設計です．まず，どのようにしてスケーラブルなスループットを達成するのかについて考えました．スループットを上げるには，動作周波数を上げるかデータを並列に処理するかが考えられます．

ここで，マイクロプロセッサなどで最近はやりの「マルチコア」というキーワードが浮かびました．深いパイプラインを切るのではなく，演算器コアを複数並べることで高性能を発揮するアプローチです．今回設計したFFT回路は，メモリ＋FFT演算器というマイクロプロセッサに近い構成であり，容易にマルチコア化ができそうです．しかも，FFTではメモリに対する演算器の規模が小さく，FFTコアを増加させてもそれほど回路規模が増えないと考えられます．

そこで，先に設計したコンパクトなFFTコアをスケーラブルに増加させるという方針で，フレキシブルなFFTフレームワークを作ることにしました．

● マルチコア回路の全体構成

マルチコア構成の概念図を**図7**に示します．並列度Pの場合，メモリがP分割されてP個の演算器に割り当てられ，それぞれがメモリ分散型マルチプロセッサのように動作します．各コアはそれぞれのローカル・メモリへアクセス可能であり，ほかのコアのメモリにもアクセスすることができます．これにより，FFTの演算時間を$1/P$に短縮することができます．

本設計では，並列度Pを指定することにより，所望のコア数のFFT回路を生成できる環境を作成しました．

● マルチコア並列動作

マルチコアによりFFTを行う動作を説明します．例として並列度$P=2$の時のFFT回路を**図8**に，動作のタイミング・チャートを**図9**に示します．

まず，データ入力状態DATAINでは64クロックかけてP個のメモリにデータを取り込みます．メモリが分散しているので，始めの$64/P$クロックでメモリ1に，次の$64/P$クロックでメモリ2に，というように順番にメモリにデータを格納します．

DATAIN状態の最後からステージ1のFFT処理を開始し，必要なデータが各コアにロードします．この際，$P=2$ではコア1は$\{x_0, x_{16}, x_{32}, x_{48}\}$，$\{x_1, x_{17}, x_{33}, x_{49}\}$，…のデータを，コア2は$\{x_8, x_{24}, x_{40}, x_{56}\}$，$\{x_9, x_{25}, x_{41}, x_{57}\}$，…のデータをそれぞれ担当し，2並列でFFTの演算を行います．しかしコア1が必要とするデータのうち，$\{x_0, x_{16}\}$はメモリ1に存在し，$\{x_{32}, x_{48}\}$はメモリ2に存在します．同様にコア2が必要とするデータのうち，$\{x_1, x_{17}\}$はメモリ1に，$\{x_{33}, x_{49}\}$はメモリ2に存在するため，このままではメモリ・アクセスの競合が起こり並列に処理できません．

そこでコア1のデータ・アクセス順を$\{x_{32}, x_{48}, x_0, x_8\}$のように変更し，先にほかのコアのメモリからデータをロードすることにしました．これに伴いバタフライ演算器

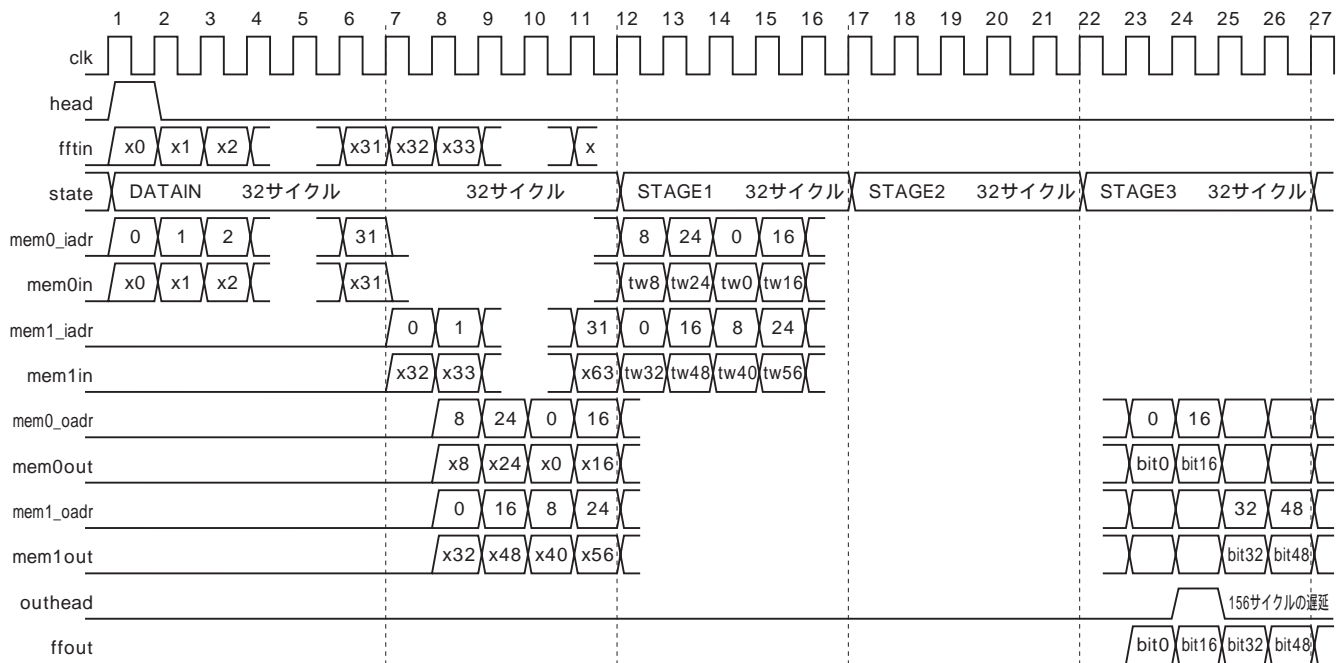



図9 2コア動作時のタイミング・チャート
レイテンシは160サイクル．

と回転因子乗算の制御を若干変更する必要が生じますが，こうすることにより二つのコアが競合なくメモリ・アクセスすることが可能となります．ステージ2以降は必要なデータが$\{x_0, x_4, x_8, x_{12}\}$のようにローカル・メモリ内に存在するため，各コアが独立してFFTを行えます．並列度Pを変えた場合にも同様にデータ処理の順番をずらすことにより，メモリ・アクセスの競合を回避しています．

● 性能

Pコアで並列に処理することにより64点データを処理する時間を短縮することができるため，スループットが向上します．

Pコアで64点を処理するために要するサイクル数は，データ入力のサイクル数と3ステージのFFT処理サイクル数の和，

$$64 + 3 \times 64/P \quad \cdots\cdots(6)$$

となります．例えば$P=1$，2，4の場合，それぞれ256，160，112となります．$P=2$，4では，それぞれスループットが1.6倍，約2.3倍向上することになります．

● 任意並列度RTLの生成

本設計ではPコアのFFT回路をそれぞれ独立に設計するのではなく，一つのソース・コードから複数並列度のRTLを生成できるようにしました．実現にはeRuby（embedded Ruby）[注1]を使用し，Verilog HDLコードにRubyスクリプトを埋め込む形式を取りました．メモリ容量，個数，演算器数などをすべてパラメータ化し，与えられた並列度Pに応じてRubyスクリプトにより適切なインスタンスを生成します．また，メモリ・アクセスの制御部について並列度Pに応じて配線や信号生成ができるようにしました．

最終的にはmakeコマンドにより，

```
% make P=2
```

のように並列度を指定すると，PコアFFTのRTLコード一式が生成される環境を作りました．

● 回路の評価

これまでと同様にDesign Compilerによりクリティカル・パスの遅延時間と回路規模を評価しました．合成結果を**表3**に示します．

回路規模の小さな演算器のみを増やす効果により，4コ

注1：任意のテキスト・ファイルにオブジェクト指向言語Rubyのスクリプトを記述し，その出力を埋め込むことのできるしくみ．冗長な記述を強いられるVerilog HDLのコーディングにおいては非常に重宝する．Emacsのverilog-modeと併用することでさらに効率的なコーディングが可能となる．

表3　マルチコアFFT回路の合成結果

	遅延時間［ns］	回路規模［ゲート］
仕様書の回路	4.98	62111
1コア	5.97	24708
2コア	6.38	32880
4コア	7.10	49755

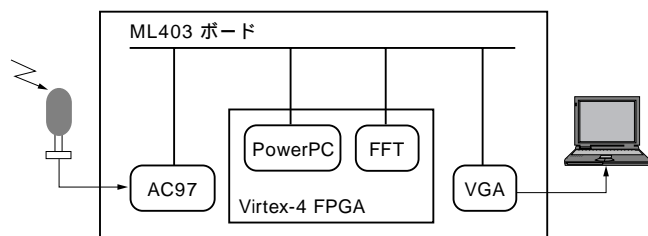



図10　スペクトラム・アナライザ・システムの概要
マイクロホンなどから入力された音声信号にFFTを行い，その結果を外部のVGA ディスプレイに表示する．

写真1　スペクトラム・アナライザの動作の様子
画面上の右下のウィンドウがアナライザ．上段に通常のパワー・スペクトルを表示している．下段はパワー・スペクトルの時間変化を表示している．

アの場合でも設計仕様書より規模が小さくなりました．なお，8コアを超えると設計仕様書より回路規模が大きくなり，あまり実用的ではない回路となるので割愛しました．

遅延時間についてはいずれも設計仕様書より増えてしまっています．今回は時間の都合上，演算器内部をきちんとパイプライン化することができませんでしたが，適切にパイプライン化することにより動作周波数を改善することが可能です．

4. FPGAへの実装

さて，回路の設計ができたので次はFPGAへの実装です．シミュレーションだけでなく，ぜひ実機で動作をさせてみたいという気持ちがあったので，FPGA実装を行いました．

まず，FFTで何をするかを考えました．比較的簡単に実装でき，かつ直観的におもしろいものを作ろうと思い，音声のスペクトラム・アナライザを制作することにしました．どうやって作ろうかと考えていたところ，ちょうど研究室にあった米国Xilinx社の「ML403 Virtex-4 FPGA ボード」が目に止まりました．このボードはFPGAのほかにPS/2，オーディオ，USB，Ethernet，VGAディスプレイなどのインターフェースが利用できる機能が搭載され，さらにFPGA（Virtex-4 FX）にはPowerPCコアが内蔵されており，1枚のボードでかなりいろいろなことができます．以前からこのボードで遊んでみたいと思っていたため，これを使わない手はありません．ということでこのボードの機能を活かしたスペクトラム・アナライザを作りました．

システムの全体像を**図10**に示します．今回使用するボード上の機能はAC97オーディオ・コーデック，VGAビデオ・コントローラ，およびFPGAに内蔵のPowerPCです．これらのモジュールはボード上のバスを介してお互い接続されています．肝心のFFTモジュールはFPGA 上に実装し，データの入出力を行うインターフェース越しにバスに接続されています．構築したスペクトラム・アナライザのシステムはマイクロホンなどから入力された音声信号にFFTを行い，その結果を外部のVGA ディスプレイに表示します．せっかくなので結果をきれいに，グラフィカルに表示したいと思い，PowerPC上でLinux OS を動作させ，X ウィンドウ・システムを用いてアナライザの画面を描画することにしました．

スペクトラム・アナライザを動作させている様子を**写真1**に示します．画面上の右下のウィンドウがアナライザです．上段に通常のパワー・スペクトルを表示しています．下段はパワー・スペクトルの時間変化を表示しており，声紋を観察することができます．

なお，本システムは沖縄で開催されたコンテスト発表会に持参し，デモを行いました．現地で正常に動いてくれるか不安でしたが，無事に動きほっとしました．

ひろもと・まさゆき
ひゅうが・ふみひこ
京都大学 大学院 情報学研究科 通信情報システム専攻 中村研究室 修士2年（当時は1年）